DQN-NECT-4: Deep Q-Networks for playing ConNECt-4

Clément Boulay

CentraleSupélec

clement.boulay@student-cs.fr

Abstract

Le Puissance-4 est un jeu dans lequel deux joueurs s'affrontent sur une grille de 7 cases de largeur sur 6 cases de hauteur. La grille est disposée verticalement et la gravité joue un rôle : les cases du bas se remplissent en premières. Un des joueurs place à chaque tour un pion de couleur jaune, tandis que l'autre joueur lui répond en plaçant sur la grille un pion rouge. Le but du jeu est d'être le premier à aligner 4 pions de sa couleur, remportant ainsi la partie. En apprentissage par renforcement (RL), une famille d'approches populaires est la famille des algorithmes dits Value-Based. Cette famille, sous-catégorie des méthodes dites Model-Free car elles n'apprennent pas de modèle de l'environnement, comprend en particulier les Deep Q-Networks. Dans ce rapport, nous souhaitons implémenter un agent utilisant un Deep Q-Network (DQN) pour apprendre une policy optimale au Puissance-4. Nous détaillerons le principe de l'algorithme ainsi que l'architecture de réseau mise en place ainsi que les méthodes d'évaluation de l'agent et nous évaluerons notre agent contre un autre agent, joueur aléatoire ou bien second agent DQN.

1. Introduction

1.1. Position du problème

L'environnement est une grille de 7 colonnes de 6 cases de hauteur. Les joueurs placent chacun leur tour un pion de leur couleur sur une des colonnes, en haut de la grille. Le jeu prend en compte la gravité, c'est-à-dire qu'un pion placé dans une colonne tombera toujours au plus bas possible dans cette colonne. Ainsi, un pion est soit superposé sur un autre dans la même colonne, soit touche le fond de la grille. Il est interdit de jouer dans une colonne possédant déjà 6 pions (on dit alors que la colonne est remplie). Le jeu termine dès lors qu'un des deux joueurs parvient à aligner au moins 4 pions de sa couleur en ligne, en colonne ou en diagonale. Si toutes les colonnes de la grille sont remplies sans que la condition de terminaison précédente ne soit atteinte, la partie est considérée comme nulle.

Le jeu de Puissance 4 est considéré comme résolu depuis 1988, d'abord par Allen puis par Allis dans [1] car il existe pour le joueur qui débute la partie un avantage stratégique qui fait que, si ce joueur joue selon une stratégie optimale (décrite dans [1]), il est certain de remporter la partie ou bien de faire match nul.

1.2. Processus de décision de Markov (MDP)

Au Puissance-4, l'espace des états possibles est l'ensemble de toutes les façons de remplir la grille de 7 par 6 avec pour chaque emplacement soit un pion jaune, un pion rouge, ou un vide. D'après [2], la taille de l'espace des états est de 4,531,985,219,092. Pour référence, au Morpion l'espace des états a une taille d'environ 5000 alors qu'aux échecs l'ordre de grandeur est supérieur à 10^{120}.

L'espace des actions est $\{0, 1, ..., 6\}$, numérotant ainsi les colonnes de 0 à 6. Toutefois, en cours de partie, si l'une des colonnes est remplie, il n'est plus possible de prendre l'action de jouer dans cette colonne, il faudra donc s'assurer de la

légalité des actions. La distribution initiale est toujours la même : la grille est vide. C'est toujours le même joueur qui commence (*player 0*). Nous préciserons dans la partie *Deep-Q-Network* les *rewards* mis en place ainsi que le *discount factor* choisi.
Enfin, l'hypothèse de Markov est vérifiée au Puissance-4, car on a bien $P(S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, ..., S_0, A_0) = P(S_{t+1}|S_t, A_t)$. Ainsi, nous travaillons bien avec un MDP.

1.3. Multiples approches mises en place par le groupe

Nous avons découpé le travail en 3. Je me suis occupé de la partie *Deep Q-Networks*, Nicolas Noblot a de son côté implémenté un Acteur-Critique et Tristan Basler a mis en place une approche *Value-Based* avec un réseau de type CNN pour tirer partie de l'agencement du jeu. Le dépôt GitHub avec notre code commun est disponible sur le lien suivant : *https://github.com/bclement1/connect-4-agent*.
Tous les rapports personnels sont également disponibles sur le dépôt.

2. Principe des Deep-Q-Networks

Le Q-Learning est une approche dite *off-policy*, c'est-à-dire qu'elle ne se repose par sur une *policy* fixée. En effet, en Q-Learning, l'agent joue une stratégie ϵ-*greedy* vis à vis de $\hat{q}$, qui est l'estimée des *Action-State values*. À terme, si les hyperparamètres ϵ et α (le *learning rate*) sont fixés correctement, l'algorithme convergera vers une valeur de $\hat{q}$ qui permet de dériver une *policy* optimale (au sens, qui maximise les *rewards*).
En Q-Learning, l'*update* de l'estimée $\hat{q}$ est :

$$\hat{q}(S_t, A_t) \leftarrow \hat{q}(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a') - \hat{q}(S_t, A_t)$$

Nous allons utiliser une variante du Q-Learning, le *Deep Q-Learning*, qui se repose sur l'idée précédente d'estimer les *Action-State values* mais l'implémente avec un réseau de neurones. Le réseau est paramétré par θ et prend en entrée l'observation courante de la grille pour retourner un vecteur de taille $|\mathcal{A}|$ (taille de l'espace des actions) qui représente $(\hat{q}(S_t, a'; \theta))_{a' \in \mathcal{A}}$. Nous pouvons ainsi mettre en place une stratégie ϵ-*greedy* par rapport à $(\hat{q}(S_t, a'; \theta))_{a' \in \mathcal{A}}$. Le réseau modifie ses poids par *backpropagation* de la façon suivante (η représente le *learning rate*) :

$$\theta \leftarrow \theta - \eta(R_{t+1} + \max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a'; \theta) - \hat{q}(S_t, a; \theta))\nabla_\theta(\hat{q}(S_t, a; \theta))$$

où a est l'action prise à l'instant t. Avec cette mise à jour, on espère que le réseau apprenne à estimer les *Action-State values* de telle façon qui l'on puisse, à convergence de ces dernières, dériver une *policy* de jeu optimale, de manière analogue au Q-Learning classique. Toutefois, il peut arriver que le réseau ne converge pas. En effet, nous sommes dans un contexte où nous combinons *Off-Policy* learning, *Value-approximation* (en l'occurence, des *Action-State values*) et *Bootstraping*, car nous utilisons une *Action-State value* estimée ($\hat{q}(S_t, a; \theta)$) pour mettre à jour cette même valeur estimée dans le réseau (de manière indirecte avec θ).

Il faudra ainsi être prudent pour fixer des valeurs d'hyperparamètres de façon à faire converger le réseau pour pouvoir dériver une *policy* optimale ensuite.

3. Implémentation

3.1. Architecture du code

Pour faciliter la convergence du réseau, nous avons implémenté une classe *ReplayBuffer*. Ce buffer retient à chaque configuration de l'agent DQN : état initial, action, état final, *reward* et booléen de terminaison. Il agrége les configurations dans sa mémoire et en retourne un échantillon de taille *batch_size* (taille de *batch* du réseau) avec sa méthode *sample* lorsque le réseau veut effectuer une descente de gradient (*minibatch*). Cela permet de stabiliser la descente de gradient et de faciliter la convergence.

Ensuite, nous avons implémenté une class MLP (Multi-Layer Perceptron) qui sera la base du réseau Q (estimation des *Action-State values*) et du réseau *target* (voir plus bas). Cette classe MLP est une simple classe PyTorch que nous ferons varier dans nos expérimentations.

Nous utiliserons deux réseaux de neurones différents pour l'apprentissage. Le réseau princpal (noté Q) est celui qui estime les *Action-State values* à chaque fois que l'on doit jouer un coup. Le réseau secondaire, ou *target network* est utilisé pour calculer les valeurs cibles des *Action-State values*. Le réseau cible est là pour stabiliser l'apprentissage, et ses poids sont mis à jour avec une périodicité réduite par rapport aux mises à jour du réseau principal. Le réseau secondaire fournit à chaque étape d'inférence du réseau principal sa propre estimation des *Action-State values*, utilisées pour calculer l'erreur et donc pour la *backpropagation*. Pour simplifier, les deux réseaux possèdent la même architecture (classe MLP), ce n'est toutefois pas une obligation dans le cas général.

En pratique, une classe DQN implémente en attribut le *ReplayBuffer* et les deux réseaux.

Notre agent DQN s'entraînera dans une fonction *train* contre un autre agent (DQN aussi ou aléatoire, voir la partie Evaluation). Une fonction de support *is_legal* prend en entrée l'état de la grille et une proposition d'action d'un des agents, et retourne un booléen indiquant si l'action est possible dans le jeu (colonnes remplies).

La boucle d'entraînement est la suivante. L'agent DQN joue par défaut en premier. À chaque fois qu'il place un pion (action forcément légale), le *buffer* s'incrémente avec la configuration courante. À chaque fois que l'agent DQN doit jouer, le Q-réseau effectue une inférence et nous suivons une stratégie ϵ-*greedy* par rapport au vecteur d'estimées de sortie (partie aléatoire pour beaucoup explorer l'espace en début d'entraînement, puis décroissance exponentielle de ce taux ensuite, contrôlée par 3 hyperparamètres). Une perte est alors calculée en comparant la sortie du réseau principal et la cible du réseau cible définie comme :

$$R_{t+1} + \gamma(\max_{a' \in \mathcal{A}} \hat{q}(s, a'; \theta_{target}))$$

Cette perte est utilisée pour effectuer la *backpropagation*. Régulièrement (voir section suivante), l'agent DQN est évalué : il joue un nombre n_{sim} de parties contre un agent aléatoire et on récupère le ratio de parties gagnées.

3.2. Hyperparamètres fixes, hyperparamètres variants

Nous avons fait le choix de fixer les hyperparamètres suivants :

- Nombre d'épisodes : 1000 pour expérience 1, 300 pour les autres (suffisant pour convergence)
- Capacité du *buffer* : 100
- bornes pour ϵ : $\epsilon_{start} = 0.9$, $\epsilon_{end} = 0.05$
- γ (*discount factor*) : 0.9

Nous ferons varier les hyperparamètres suivants, que nous pensons être les plus critiques pour l'apprentissage :

- fréquence de mise à jour du réseau cible, notée f_{target}
- taille des *batch* pour le calcul de la perte du Q-Network
- η (ou α) (*learning rate*)
- taux caractéristique de décroissance exponentiel pour ϵ, noté τ

Enfin, nous fixerons la fonction de perte en prenant une distance L2 et l'optimiseur en prenant Adam.

4. Evaluation de l'agent et performances, impact des hyperparamètres

4.1. Expériences menées

Après l'entraînement, nous évaluons une seule fois sur 100 parties notre agent DQN contre un agent aléatoire. Les tableaux suivants donnent les performances empiriques mesurées.

f_{target}	8	16	32	64
Ratio de parties gagnées (évaluation finale)	0.48	0.45	0.49	0.37

Table 1. **Expérience 1** : variation de f_{target} à $\eta = 0.1$, $batch\_size = 8$, $\tau = 1000$

η	0.1	0.05	0.01	0.001
Ratio de parties gagnées (évaluation finale)	0.56	0.4	0.42	0.51

Table 2. **Expérience 2** : variation de η à $f_{target} = 32$, $batch\_size = 8$, $\tau = 1000$

batch_size	8	16	32	64
Ratio de parties gagnées (évaluation finale)	0.41	0.46	0.4	0.43

Table 3. **Expérience 3** : variation de $batch\_size$ à $f_{target} = 32$, $\eta = 0.1$, $\tau = 1000$

τ	10	50	100	1000
Ratio de parties gagnées (évaluation finale)	0.38	0.48	0.49	0.39

Table 4. **Expérience 4** : variation de τ à $f_{target} = 32$, $\eta = 0.1$, $batch\_size = 8$

4.2. Interprétation

D'abord, on remarque qu'en mettant à jour le réseau cible trop rarement (tous les $f_{target} = 64$ coups par exemple), le réseau principal (Q-network) semble être plus en difficulté pour converger. Les performances générales pour la première expérience semblent toutefois suggérer que le réseau n'a pas réussi à apprendre suffisamment pour pouvoir battre largement le joueur aléatoire, et ce peu importe la valeur de f_{target} implémentée. Cela laisse penser qu'un autre paramètre est mal fixé, et que régler la valeur de f_{target} ne permet pas de restaurer la convergence de l'ensemble.

D'après la seconde expérience, le taux d'apprentissage η peut jouer un rôle dans la convergence. Le plus grand taux d'apprentissage, $\eta = 0.1$, semble avoir permis au réseau d'atteindre de meilleures performances que celles des autres configurations testées. Toutefois, on observe également des performances proches du 50:50 sur cette expérience, même si c'est moins le cas pour le meilleur learning rate trouvé. Ainsi, il faudrait essayer d'évaluer sur plus d'itérations d'entraînement et de test la première configuration afin de voir si elle correspond vraiment à un gain de performances.

Pour la 3ème expérience, on remarque que le batch size ne semble pas jouer de rôle déterminant pour les performances du réseau, qui sur ce cas ne semble pas avoir convergé non plus. On remarque également que le nombre de parties jouées pour l'évaluation n'est peut-être pas suffisant pour correctement estimer les performances du réseau. En effet, on retrouve, pour un batch size de 8, la configuration déjà mise en place dans l'expérience 2 (première case) qui avait alors donné un résultat plutôt positif. Ici, la valeur de 0.41 semble suggérer que cette configuration n'a en fait pas vraiment converger.

Enfin, pour la dernière expérience, les commentaires ci-dessus semblent s'appliquer également. On semble tout de même déceler qu'une valeur de $\tau = 50$ ou $\tau = 100$ paraît un bon choix pour maximiser la performance de l'agent.

Il me paraît étonnant de voir que varier les paramètres du modèle ne permette pas faire converger ce dernier et d'atteindre des performances largement meilleures que l'agent aléatoire. Deux facteurs peuvent causer cette non-convergence. D'abord, le nombre d'épisodes peut être trop faible. En effet, on a lancé de 300 à 1000 épisodes par essai, ce qui peut être trop peu pour atteindre la convergence. Toutefois, les courbes d'apprentissage visibles dans le code suggèrent que l'apprentissage semble se déclencher rapidement puis s'arrêter du fait de valeurs de pertes nulles. Cela nous suggère (deuxième hypothèse) que le problème vient plutôt de la forme des rewards du problème, qui sont *sparse*. En effet, la plupart des coups joués apportent à l'agent un reward de 0, ce qui ne le pénalise pas. Seule une victoire (+1) ou une défaite (-1) amènent un reward non-nul à l'agent, qui peut alors tirer des conclusions. On pourrait mettre en place une stratégie de rewards qui ne soit pas *sparse* pour assurer la convergence de l'agent. Cette approche a été mise en place dans d'autres travaux du groupe (en particulier dans le cadre de notre approche Acteur-Critique) et montre des résultats encourageants.

5. Conclusion, généralisation à d'autres problèmes et extensions possibles

Pourrait-on généraliser notre agent DQN à d'autres jeux ? Notre approche repose sur l'estimation des valeurs *Action-State*, peut en théorie être transposée à tout autre jeu pour lequel l'espace des états-actions est discret et n'est pas trop grand, et pour lequel les actions sont déterministes. Ainsi, on s'attend à ce que la mise en place pour le jeu des échecs et le jeu de Go (discrets, grand nombre d'états, déterministes) soit possible mais délicate. De plus, aux échecs et au Go comme au Puissance-4, l'agent possède une information parfaite (la grille ou le plateau de jeu). Pour un jeu continu, temps réel et à information incomplète comme Starcraft, l'espace des états devrait être discrétisé pour permettre d'implémenter notre approche, car cela serait très difficile sinon impossible pour un espace continu, de grande dimension et dynamique tel que celui utilisé. De plus, au Puissance 4, les actions possibles sont limitées, ce qui est relativement le cas aux échecs et au Go, mais moins dans Starcraft, pour lequel l'information de l'état courant est déstructurée (ce n'est pas une simple grille) et difficile à extraire. Cela rend l'estimation de $\hat{q}(S_t, A_t)$ particulièrment complexe.

Enfin, avec plus de temps et de ressources de calcul, nous aurions voulu faire varier les hyperparamètres tester sur des plus grandes régions afin de voir si nos interprétations tiennent lorsque les valeurs prises sont plus extrêmes. Nous aurions également voulu mettre en place deux architectures différentes pour le réseau principal et le réseau cible, possiblement plus profondes et complexes, et jouer un plus grand nombre de parties pour voir s'il était possible d'atteindre une convergence, et si oui connaître les performances correspondantes.

References

[1] V. Allis, "A knowledge-based approach of connect-four," 1988. [Online]. Available: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf.

[2] "Connect 4," *Wikipédia*, [Online]. Available: https://en.wikipedia.org/wiki/Connect_Four.